UCRL-JC-116038
Rev. 1

Evaluating Software for Safety Systems in Nuclear Power Plants

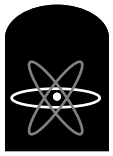
Prepared by

J. Dennis Lawrence, Warren L. Persons, and G. Gary Preckshot
Computer Safety & Reliability Group
Lawrence Livermore National Laboratory

John Gallagher
Human Instrumentation and Controls Branch
Office of Nuclear Reactor Regulation
U. S. Nuclear Regulatory Commission

Prepared for

U.S. Nuclear Regulatory Commission



FESSP
Fission Energy and Systems Safety Program

Lawrence Livermore National Laboratory

Disclaimer

This work was supported by the United States Nuclear Regulatory Commission under a Memorandum of Understanding with the United States Department of Energy, and performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Evaluating Software for Safety Systems in Nuclear Power Plants

J. Dennis Lawrence, Warren L. Persons, and G. Gary Preckshot
Lawrence Livermore National Laboratory

John Gallagher
U. S. Nuclear Regulatory Commission

Manuscript Date: April 8, 1994

1. Background

Commercial nuclear reactor technology has been in existence for more than forty years. At the end of 1992, 419 commercial reactors were in operation, with an additional 59 under construction [1]. (The U.S. numbers are 112 and 6, respectively.) Compared with other technologies, reactor technology has had an excellent safety record, and there is considerable interest in continuing and improving upon that record.

Existing reactors generally use separate instrumentation and control (I&C) systems for control and for protection—indeed, this is mandated by law in the United States. The protection systems are relied upon to initiate required safety functions following abnormal transient or accident conditions. Both types of I&C systems have historically used analog systems coupled with either electro-mechanical relay logic or hardwired solid-state logic as the basic technology.

In the United States, most reactors have been designed individually, built by a company specializing in reactor technology, owned and operated by an electric power utility, and regulated by the U.S. Nuclear Regulatory Commission (NRC). Regulation is limited to assuring public safety to the maximum extent possible. Regulation has included the granting of a construction permit before construction of a reactor begins, granting of an operating license after construction, and inspections during construction and operation. Public hearings are held before issuing the construction permit and operating license.

A number of changes are taking place both in I&C technology and in regulation. New reactor designs will permit reuse of a design at multiple sites. At the same time, I&C technology is moving toward the use of digital computers. Moreover, U.S. law has expanded the number of methods of licensing. In the future, the NRC may issue a design-standardized certification for a particular reactor design. A utility wishing to build a reactor will obtain a combined construction and operating license, and may use any certified design. Once built, the plant can begin operation without additional mandatory public hearings, but not until completion of certification inspections, tests and analyses against approved acceptance criteria.

As part of the design certification, the applicant for a standard design certification will submit a set of inspections, tests, analyses and acceptance criteria (ITAAC) such that if the inspections, tests and analyses are carried out, and they meet the acceptance criteria, the plant as constructed is considered to be safe. The utility/license applicant will carry out the ITAAC, and the NRC will ensure that it has been properly implemented using an audit process.

A design certification will be in force for fifteen years and can be renewed at the end of that period. A plant constructed using such a design might well operate for forty years, with the possibility of plant license renewal for an additional twenty years. The result is that a plant built to a design certified today may still be operating nearly a century from now. Thus, it is imperative that the design be safe!

The safety of the design is considered to be a relatively routine determination for many aspects of the nuclear power plant—i.e., concrete and steel technology is reasonably

well understood, fairly stable, and not likely to change over the next fifteen years. However, a potential area of concern is computer-based I&C systems that control the plant and initiate the plant protection systems, both because their use in U.S. nuclear power plants is relatively new and because the technology is changing rapidly. The computer-based safety systems are of particular concern to the NRC.

The Lawrence Livermore National Laboratory (LLNL) has been working on aspects of nuclear power technology for nearly half a century, both for the Department of Energy and its predecessors, and for the NRC. This includes work on seismic analysis, transportation of nuclear materials, nuclear physics, fusion power, long-term waste storage, isotope separation, radiation health issues, digital I&C issues, and the like.

Much of the experimental work in physics conducted at LLNL requires the use of automated control systems, so the Laboratory has been building some of the world's most advanced digital control systems for decades. Examples include the Magnetic Fusion Test Facility, the Nova facility and the Atomic Vapor Laser Isotope Separation facility.

In 1991, LLNL was asked by the NRC to provide technical assistance in various aspects of computer technology that apply to computer-based reactor protection systems. This has involved reviewing safety aspects of new reactor designs and providing technical advice on the use of computer technology in systems important to reactor safety. The latter includes determining and documenting proposed regulatory criteria for the NRC on the development and implementation of digital computer safety systems. These aspects include data communications, formal methods, testing, software hazards analysis, verification and validation, computer security, performance, software complexity, and others. One topic—software reliability and safety—is the subject of this paper.

2. Methodology

The purpose of the software reliability task has been for LLNL to assist the NRC in understanding the state of the art in assessing the reliability of the software for a computer-based reactor protection system. Three separate activities were carried out: assess the status of national and international standards that relate to software reliability and safety, obtain advice from technical experts in software reliability and safety, and assess the best current practice in industry. The first activity was reported in [2], so is not discussed further here.

2.1. Technical Experts

In FY 1992, LLNL sponsored a small workshop in which four leading software experts were brought together with members of the NRC staff to give their opinions on methods and techniques for the development of safety-critical software, and to discuss the reliability and safety of reactor software. Experts were chosen based on reputation, technical expertise, availability, and interest in the NRC's problem in order to achieve the shared goal of obtaining a variety of viewpoints. They were:

- Ricky Butler, NASA Langley.
- Nancy Leveson, University of California at Irvine.[1]
- Bev Littlewood, City University of London.
- John Rushby, Stanford Research Institute.

The workshop was held in July 1992 in San Diego, California, and lasted a day and a half. A report on the workshop is available from LLNL [3].

2.2. Commercial Practice

In FY 1993, the investigation was expanded to study the best contemporary industry practices. Specific organizations within three corporations were chosen based on reputation, availability, and willingness to participate. They were:

- Computer Sciences Corporation (CSC), Systems Engineering Division.
- International Business Machines, Federal Systems Company (IBM/FSC), Space Shuttle Project.[2]
- TRW, Ballistic Missile Defense Division; Army WWMCCS Information System Project; and Universal Network Architecture Services Project. (Three projects within TRW were included in the study.)

Supplementary conversations were held with representatives of the NASA Software Engineering Laboratory and the American Institute of Aeronautics and Astronautics (AIAA) Software Reliability Project.

Interviews and discussions with the three organizations were directed at "what works." Specifically, the discussions began with the following questions and expanded from there:

1. What are the most important obstacles to producing highly reliable software for use in safety-related applications?
2. Why is your company successful at producing highly reliable software?
3. What evidence exists that the methods work?
4. What are the most important lessons your company has learned about producing highly reliable software?

The results of the interviews and discussions were combined into a set of principles which were termed "design factors." This list was then discussed with a fourth company as a validation check. This company, which prefers to remain anonymous, is also highly regarded for the excellence of its software, but represents a very different

[1] Now at the University of Washington.
[2] Now part of the Loral Company.

segment of industry from that of the three primary organizations. The results of this investigation have been documented and are available from LLNL [4].

3. Common Principles

Although the areas of emphasis among the three sources of information (standards, experts and organizations) tend to be quite different, no substantial areas of disagreement were found, providing considerable confidence in the results of the study as applied to reactor protection systems and other safety-critical applications. Some broad, shared principles are given in this section. More detailed recommendations can be found in Section 4.

3.1. Safety Is a System Problem

This principle is generally accepted, but the implications for designers and regulators deserve some comment. Reactor protection systems are designed using the principles of diversity and defense-in-depth. The first implies that several different physical properties of the reactor are monitored for possible deviations from normality. Different logical combinations of sensed parameters are used to indicate problems, and different physical methods of response are used to prevent or mitigate failures. Thus, during an event or accident, a number of diverse signals that follow diverse paths through the equipment will cause reactor trip, core cooling, or containment isolation.

The second principle means that in typical reactor I&C designs there are multiple echelons of defense: the control and monitoring system is designed to keep the reactor out of trouble in the first place; the reactor trip system is designed to shut the reactor down when the operating envelope is exceeded; the safety parameter display system and manual controls are designed to provide enough information and capability for operator response; and the engineered safety features actuation system will actuate the emergency core cooling, containment isolation, and cooling systems to cool the core and maintain containment integrity when required. At the physical barrier level, the core cladding is designed to prevent the release of radioactive material to the coolant; the pressure vessel and reactor coolant pressure boundary are designed to prevent the release of the coolant; and the containment is designed to prevent the release of radioactive material if the reactor coolant pressure boundary fails.

This discussion has some serious implications for software design. A balance should exist between the general architecture of the protection system and the software contained within it, so that functions are carried out at the most effective location within that architecture. For example, it is required by law that reactor protection systems have no single point of failure, so that multiple, redundant computer processors are required which must be resistant to simultaneous failure. Since computer systems can fail either because of hardware or software failures, a serious new concern (common-mode software failure) now exists where none existed before. A system-level response to this concern is to use the system architecture to provide defense-in-depth (instead of attempting some form of n-version programming). An alternative is to ensure sufficient diversity among programs so that several programs do not execute almost-identical

sequences of instructions on identical inputs, as faults in the programs can lead to common-mode failures within redundant sets of digital processors.

3.2. Risk-Based Design

No company has unlimited resources—money, time, and (perhaps most important) highly qualified talent are all finite. It is also generally true that some portions of a software system have higher risk (in terms of the consequences of failure) than others. Indeed, it is frequently good practice to design software so that those portions whose failure can lead to the most adverse consequences are small, isolated and confined. Such software can be termed a "safety kernel."

Risk-based design means that a formal risk analysis [5] is performed to identify the most risky portions of the software, to design both the overall protection system and its software to isolate and confine risk, and to concentrate resources on the areas of highest risk.

3.3. An Understanding of Complexity

The term "complexity" may be applied to many aspects of the software development process and products, and some of these are naturally more important than others. For the purposes of this paper, three types of complexity are recognized: functional complexity, structural (or design) complexity, and code complexity, with the caveat that none of the three have absolute definitions.

Functional complexity refers to the difficulty of the problem being solved—the functions that the software is expected to provide, and the interactions among those functions. This is very difficult to measure, so may receive insufficient attention.

Structural (or design) complexity refers to the complexity of the software architecture and design. The design must of course be sufficient to cover all the functional requirements, but should not be more complex than necessary.

Code complexity refers to the internal complexity of the code modules. This is the easiest to measure, so generally receives the largest share of attention.

The fundamental issue is that complexity becomes a problem when it decreases the understandability or the verifiability of the computer system. No complexity aspect can be ignored, since concentrating on a single aspect just encourages designers to shift complexity elsewhere. For example, demanding that code modules have cyclomatic complexity metric less than ten is likely to result in an overly complex structure of modules, each of which has cyclomatic complexity metric less than ten.

3.4. Commitment to Quality

Perhaps the most outstanding shared characteristic of the organizations interviewed was an understanding of, and a top-to-bottom commitment to, the production of quality software. All three organizations demonstrated in various ways their pride in the quality of the software they build, and reward their people accordingly.

Quality did not come easily or painlessly. Current levels of achievement have taken a decade or more to reach. Mistakes have been made, but the attitude is "Why did we make the mistake?" and "How can we avoid it in the future?" rather than "Who shall we blame this time?"

Finally, none of the three organizations is satisfied with its current status. Instead, continuous deliberate planned quality improvement appears to have become an integral part of their corporate cultures.

3.5. Multi-Level Evaluation

It is well accepted, at least among software practitioners, that there is no "silver bullet" [6] that can be used to magically produce reliable software. There is likewise no silver bullet for regulators to use in satisfying themselves that a software-based protection system is safe.

One of the principal results from the work at LLNL is the belief that assessing the safety of any software system involves at least three levels of evaluation: the development company and its culture, the specific development process for the product under review, and the software and documentary products of the development process.

This approach attempts to assure safety through the cumulative weight of diverse forms of evidence obtained primarily during the development process. It provides a qualitative argument for safety, in contrast to a quantitative statistical assessment of the safety of the software system, such as the probability of failures per demand (f/d). This appears to be the best that can be done with the current state of software technology. No guarantee can be given of absolute safety—but that is true of all technologies.

Since much of the evidence available is qualitative in nature, it may be quite difficult to calculate an accurate measure of safety, although crude bounds may be possible. That is, it may not be possible to calculate the probability of an accident due to software failure to within a factor of two, but it may be possible to show that the probability is less than, say, 10^{-3} or 10^{-4} f/d, and add a well-founded belief through the weight of evidence that the probability is much lower than 10^{-3} f/d. This may be satisfactory if backed up by "an intellectually convincing argument."[3]

4. Design Factors

Given the failure of magic, the lack of "silver bullets," and the limitations of quantitative reasoning, confidence in the quality of any software product can only be acquired through years of proven successful use or a preponderance of a variety of evidence. The first alternative is unattractive (and most likely unattainable) when safety is important and is unlikely to exist in the rapidly changing computer industry. This leaves the challenging alternative of evaluating multiple factors relating to the software development organization, design process, and software product. Success in other

[3] Our thanks to John Rushby for this phrase.

assessment arenas, such as TickIT [7] in the United Kingdom, provides additional confidence in this approach.

The research reported here has identified a number of design factors that can serve as the basis for a safety assessment. Supporting evidence comes from standards, expert opinion as given in the workshop and the technical literature, and the best current industry practice. Positive and negative design factors have been identified. They are listed here under five headings: general, management, process, product, and negative factors.

4.1. General Design Factors

The factors listed here relate generally to the development of safety-critical software.

- All levels of the development organization demonstrate a commitment to quality.

- There is longevity in personnel, policy and process which provides stability in organizational culture and continuous improvement over many years.

- Configuration management is implemented rigorously and used extensively during all life cycle phases and for all life cycle products, including test, verification and validation (V&V) and quality assurance (QA) products.

- Testing, verification and validation, and quality assurance activities are independent from the development activities.

- The test team is involved from the very beginning of software development, to assure that the requirements are testable, to assure that the software to be developed employs design features that are within the capability of the test tools, and to provide input to the test cases, including test data and operational profiles.

- The test team and the V&V team include subject area specialists as well as software experts.

- Reviews, walkthroughs, and inspections are used at all stages of development and for all products, including testing, V&V, and QA products.

- Continuous process improvement is a corporate way of life.

- The organization has long-term experience developing safety-critical applications.

- Software is one of the main businesses of the organization, and top management understands the process of software development.

- The organization has a goal of defect-free software, and spends the energy and resources to come as close to realizing the goal as humanly possible.

- Quality is built in. It is not possible to "test in" quality. Instead, quality must be designed into the product and that fact should be demonstrated by testing, V&V, and QA activities.

4.2. Management Design Factors

The factors listed here relate to the organization's management of software development efforts. The underlying principle is an intelligent, deliberate commitment to quality.

4.2.1. General Management Design Factors

- Vendors, products, and services obtained from others are certified to the level required to support safety-critical applications.

- The organization monitors, understands, and adapts to the changing environment in the computer industry.

4.2.2. Process Control Design Factors

- The organization has a well-defined, detailed software development process model.

- The software development process is stabilized through measurement, feedback, and gradual improvement.

- Process improvement occurs in two stages: a stabilization stage to provide a firm foundation for measurement of the effects of one (or at most a few) changes, which in turn is followed by measurement of the stabilized process outputs.

- Process data which is appropriate to the maturity of the organization is collected, understood, and used as the basis for process improvement.

4.2.3. Management Action Design Factors

- The organization's reward structure matches its commitment to quality, rather than undercutting it by primarily rewarding the meeting of budget and schedule targets.

- The organization's management actually uses its process models.

- Management predicts the effect of process changes through its process model, and measures results to obtain feedback on the actual effect of changes. In the language of control theory, the measurement lag would make an extremely sluggish system or an unstable one without the anticipation provided by predictions.

- Management has a successful track record planning, allocating resources, and meeting schedules within cost and quality constraints.

- Management actively identifies, analyzes, and manages business, technical and safety risks.

- Management abandons methods that do not work. (This may seem obvious, but abandoning an inappropriate "improvement" can be a career risk in some organizations.)

- Management ensures the planning, production, and control of documentation. Because documentation tends to be neglected under stress, this serves as a sensitive indicator of management commitment.

- Management invites external review.

- Management understands that improvement takes time—typically, it takes about two years for a process change to become stabilized. Impatience and the search for short-term gains is not a virtue.

4.2.4. Personnel Design Factors

- Programming skill is not enough when safety is critical—some people must be skilled in the problem domain.

- It is well recognized that the single greatest factor in assuring quality software is the knowledge, skill, and intellectual ability of the technical staff.

- The greatest single obstacle to producing high-quality software is inaccurate interpersonal communications. As a corollary, project teams should be kept small (6–8 people).

4.3. Process Design Factors

The factors listed here relate to both management and technical aspects of the software development process. The underlying principle is a deliberate planned approach to software development.

4.3.1. General Process Design Factors

- Reviews, walkthroughs, and inspections are used at all stages of development and for all products, including code, development documents, V&V products, and QA products.

- Resource investments are targeted throughout the process life cycle according to safety and reliability requirements.

4.3.2. Technical Planning Design Factors

- An appropriate life cycle model is deliberately chosen and used.

- Life cycle activities are chosen to promote early detection of errors.

- The software architecture is chosen to isolate and confine risks, and the software elements are managed appropriately according to risk.

4.3.3. Requirements Specification Design Factors

- Requirements are stable. For a reactor protection system, unstable requirements reflect inadequate system design.

- Requirements are analyzed to understand their implications: to detect inconsistencies, unneeded but expensive specifications, over-specification, and requirements that may be extremely difficult or impossible to fulfill; and to ensure that the requirements are correctly translated from application-specific terminology to software-specific terminology.

- Requirements are validated against the system design and the safety analysis report.

- Prototyping and simulation are used in appropriate ways to refine requirements, test design approaches, and demonstrate system performance.

- Requirements are written to be testable.

- The requirements analyst prepares and documents a well-constructed argument as to why the requirements are correct and complete.

4.3.4. Design Specification Design Factors

- Safety-critical software components are identified early in the design process so that sufficient resources can be directed to them.

- A design philosophy suitable for safety-critical software is used. In particular, "risky" practices are avoided.

- An appropriate level of complexity is defined for the product and documented practices are followed that control this complexity.

- The product is designed to permit easy understanding, testing, and verification.

- The designer presents and documents a well-constructed argument as to why the design is correct, complete, reliable, and safe.

4.3.5. Software Quality Assurance Design Factors

- The software quality assurance effort is organizationally independent of the development organization.

- Quality is built in. It cannot easily be retrofitted, and cannot be "tested in."

- Defect tracking is taken seriously, is carried out uniformly and consistently, and is statistically valid.

- The root causes of defects are determined, and appropriate corrective actions are initiated to reduce the probability of similar errors in the future.

4.3.6. Safety Design Factors

- Hazards analysis is used as part of the development process. Hazards can be introduced by the selection of design approaches, certain hardware, software tools, or the use of software itself as a solution to a safety problem.

- System diversity is used to improve reliability and safety, and the software is designed to be an appropriate part of the system.

- The safety system of which software is a part may be circumvented, turned off, or driven into failure by operator actions. Neither the system nor the software can be expected to prevent these problems. These implications are factored into the software design.

- Reliability in the order of 10^{-5} to 10^{-6} failures per demand[4] cannot be assured by any known method of implementation or testing. The software portion of the system must either be designed in such a way that these levels of reliability are not required, or the inability to quantify reliability at these levels must be accepted.

4.3.7. Testing Design Factors

- Testing is carried out at multiple levels: unit, subsystem, and system.

- The software testing effort is organizationally independent of the development organization.

- Testing has its own life cycle, which is planned, designed, implemented, and followed in parallel to the development life cycle.

- Testing goals do not exceed the current practical limitation of about 10^{-3} to 10^{-4} failures per demand.

4.3.8. Verification and Validation Design Factors

- The software verification and validation effort is organizationally independent of the development organization.

- Requirements validation is performed.

- V&V is planned early in the life cycle, and the V&V plan is peer-reviewed.

- The software product is designed to permit effective verification and validation.

4.4. Product Design Factors

[4] Roughly equivalent to 10^{-8} to 10^{-9} failures per year in continuously operating vehicles such as aircraft.

The half dozen factors listed below are aimed at creating a predictable, verifiable software system. The underlying principles are simplicity and determinism. The presence of these factors is considered a positive indication of lowered complexity or easier error detection.

- *No interrupts.* The use of interrupts, beyond a simple clock interrupt, is considered a higher-risk implementation method because of the extra care required to ensure correct synchronization between interrupt code and interrupted code, and to ensure that interrupted code is correctly resumed.

- *No multi-tasking.* Multi-tasking requires context switching and task management in addition to the complications attendant upon using interrupts.

- *Simple loop design.* A single loop program structure is the simplest program organization capable of continuous operation.

- *Deterministic predictable timing.* Evidence should exist that software product timing is a predictable function of load, and that load is limited by design.

- *No pointers.* The use of explicit pointers (addresses) of data has been taken by some as a risky practice. The potential exists for errors in programmer-directed address arithmetic which would not exist if named variables were used and the addresses were computed automatically by compiler.

- *Strong data typing.* Data typing permits compilers to detect data misuse errors (e.g., using an integer as if it were a floating point number). This class of error represents a significant proportion of all errors made, and strong data typing with good compilers almost eliminates it.

4.5. Negative Factors

The presence of any of these factors provides ample justification for caution and concern by the regulator. A more thorough investigation may well be in order.

- *High turnover.* The most obvious implication of high turnover is that building a team of high-quality people with a team memory is impossible. Less obvious is the fact that high turnover is a comment (by software engineers and managers who leave) on the competence of management that is left behind. It should not be ignored.

- *Projects are schedule-driven.* The first victims of a missed deadline are usually quality assurance and documentation. The next victim is the testing program. A "deliver at all costs" mentality is cause for concern.

- *Organizational process history is short or lacking.* The research results are explicit about the length of time it takes to build a quality software organization.

- *Management cannot enforce stable requirements.* Stable and complete requirements are necessary for quality software products, but the role of management in ensuring this cannot be emphasized enough. Not only must management demand that requirements

be locked down, but management itself must not be the source of requirements thrashing. Requirements instability and weak management control are indicators of potential failures.

- *Management has a record of failing to meet predicted cost, schedule, and quality goals for products.* This is typically an indication of management by chaos or paradoxically, schedule-driven rather than quality-driven development. Schedule- and budget-driven development schemes often fail to meet delivery schedules because of product non-performance problems. Something is delivered, but it is not the contracted-for product. A record of failing to meet cost, schedule, and quality goals should be taken seriously as an indicator of deeper troubles.

- *The organization fails to track errors and defect causes.* An organization's record of errors, causes, and corrective actions is its won-loss track record. No record, or a haphazard record, should be taken by default as meaning a bad record.

- *The effort is underfunded.* Several of the companies interviewed suggested that most large government software contracts are underfunded by at least a factor of two with the expectation that more funds can be obtained later by litigation, contract expansion, or cost overrun procedures. Whatever the reason, underfunding results in staff transients and failures to carry out "non-essential" but vital activities such as quality assurance, documentation, and V&V. While it may be difficult for an outside reviewer to estimate what a correct funding profile should be, this negative factor is very real.

- *The organization exhibits "kill the messenger" syndrome.* Several companies interviewed emphasized the need for administrative procedures by which bearers of bad tidings could unburden themselves without jeopardizing their careers. They noted that organizations without these mechanisms were often the last to know about internal problems.

5. Experiences Developing Highly Reliable Software

The companies interviewed were very generous in supplying data on their experiences in developing highly reliable software, a portion of which is presented below. The data is from specific projects in specific companies, and may or may not be applicable to other projects within the companies or to other companies. Interpretation of the data is the responsibility of the authors.

Figure 1, from Computer Sciences Corporation, shows the percentage of different classes of errors detected during one project by different methods of testing. Five levels of testing were used, referred to as Level 0 through Level 4. Errors detected were placed in six classes.

Level 0 testing consisted of peer reviews, design walkthroughs, code walkthroughs, and documentation walkthroughs. Level 1 testing consisted of unit testing, level 2 of component testing, level 3 of program and subsystem testing, and level 4 of complete system testing.

The column labeled "total system errors" shows the approximate cumulative percent of all errors discovered by the end of each level of testing. The remaining six columns show the approximate percent of each class of error found by each level of testing. For example, 70% of design errors were found during level 0 testing, while 50% of interface errors were found during level 2 testing.

The information given in this figure plus other information from CSC that is discussed in reference [4] support the statements made in 4.3.7. above.

Figure 2 provides a history of errors discovered by the IBM/FSC on-board shuttle project. IBM/FSC employs a rigorous method of error tracking. Each error discovered during development or operation is traced back to the time when the error was placed in the code, and counted against that code release. Since a great deal of code is reused from one shuttle flight to another, this tracking enables FSC to determine the exact time the error was inserted into the code, and to determine the reason this occurred.

Errors are classed as early detection, independent verification, and product. Early detection errors are those detected before a new system build occurs. Independent verification errors are those errors detected after a system build occurs but before the new build is delivered to the customer. Product errors are those errors detected by the customer.

Figure 2 must be read carefully. Approximately one product is released per year, and errors are counted against that release as long as they can be traced back to the release. This means, for example, that a version released in 1985 has been counting product errors for 8–9 years, while a version released in 1993 has barely begun counting product errors. The result is that the product error curve may be somewhat biased. However, on-board shuttle systems within the past five years have shown virtually no errors discovered after first flight usage. The number of early detection errors in each release and the number of process errors in each release do not have this bias since, by definition, these error counts do not change after release. The information presented in Figure 2 together with the discussion of other information obtained from the IBM/FSC on-board shuttle project provided much of the basis for the Management Design Factors in Section 4.2. It is important to point out that while the total on-board software program is on the order of 500 KLOC the individual builds (product releases) were usually in the range of 5 to 50 KLOC, which is essentially equivalent to the range for individual software systems for nuclear power plants.

Figure 3, also from IBM/FSC, provides one indication of the cost of high reliability. The curve is derived from historical data on the shuttle project, and shows that the amount of effort which must be devoted specifically to the independent V&V (IV&V) effort for a project can be estimated from the criticality of the project, measured by the required product error rate in terms of errors per thousand source lines of code (KSLOC). "Ordinary" products can be produced by devoting approximately 10% of the development cost to IV&V, resulting in about one error per KLOC in the delivered code. Reducing error rates below this requires much larger investments in money, time, and

labor. To achieve a delivered error rate of 0.1/KLOC, approximately 40% of the project budget must be spent on IV&V.

The information presented in this figure plus discussions with all three organizations provided valuable input into the importance of Management Action Design Factors with respect to planning for and allocating resources for the development of safety-critical software.

6. A Possible Approach to Assuring Safe Software

Figure 4 resulted from a discussion between the authors and Victor Basili, University of Maryland, in which the general theme of the discussion was the intellectual activities involved in the development of software, in particular reading and the need for some form of reading protocols for the different roles in software development. The figure is considered speculative. In this approach, each phase of a life cycle would be thought of as containing two parts: a construction part and an analysis part. The construction portion consists of the usual life cycle activities of requirements, design, code, installation, and operation. The analysis portion consists of all activities directed to ensuring the correctness of the construction activities.

Analysis can be error-based or phase-based. The figure shows the analysis activity that might occur after the requirements specification has been constructed. There would be a similar activity after all other life cycle phases. The figure shows two types of analysis taking place on the specification. These analyses may be of any type; in the example, it is assumed that models will be developed. The first type of analysis looks for errors. The other form of analysis is based on the specific life cycle phase under consideration. For example, an initial test plan can be written, and an initial attempt at a software design can be produced. If these can be done in a reasonable manner, confidence in the requirements specification is increased.

In the approach suggested here, an audit team could examine the specification and the various analyses to be sure they are done correctly. One set of products of the audit will be risk analyses. Three are suggested here: financial, schedule, and technology. These risk analyses will be kept, and used as input to the audit of the next phase; that audit will pay special attention to how the risks were handled by the construction team. In this example, suppose that the audit of the requirements specification determines that there is a significant risk that the schedule cannot be met. This will be documented. After the design specification has been completed and analyzed, the design audit team will want to know what was done to address this schedule risk. An additional risk analysis will be done on the design; perhaps the design is complex, requiring new workstation technology. This will be documented, and used as input to the next phase audit, and so on.

7. Conclusions

It is certainly true that unsafe software has been created, and will probably be created in the future. It is equally true that very good software has been created, and will continue to be created in the future. As with any technology controlling a safety-critical

application, software has contributed to death and injury [8], but it is somewhat surprising how rare this has been. Why?

The thesis presented in this paper is that much of the difference between success and failure can be attributed to the knowledge, understanding, intelligence, and care of the individuals and companies involved in the development of safety-critical software. By combining the best from theory and practice it is possible to isolate a number of factors that distinguish the good from the bad.

There is a great deal of emphasis in the literature, and in conferences such as this one, on the negative consequences of software failures in safety-critical applications, and this is appropriate. However, there are also several balancing, positive factors which deserve equal emphasis. In particular, software does not wear out, potentially can be used to identify and compensate for hardware failures, and potentially can provide much greater control to operators during unexpected events. These factors should be carefully evaluated on a case-by-case basis to determine the suitability of software in each plant application.

The challenge faced by software developers is to use software safely to increase the reliability of the application, while the challenge for the NRC evaluator is to assure that this is done. The research conducted as part of this contractual effort suggests that convincing evidence can be obtained in practice that reliable safety-critical software is being or has been developed. However, neither the development of such software nor the effort required to certify it for safety-critical usage is easy.

References

1. "On a plateau after 50 years," *Nuclear Engineering International* **38**, #467 (June 1993), 18–24.

2. Debra Sparkman, "Standards and practices for reliable safety-related software systems," Proc. Third International Symposium on Software Reliability Engineering, IEEE Computer Society (October 1993), 318–328.

3. J. Dennis Lawrence, "Workshop on developing safe software: final report," UCRL-ID-113438, Lawrence Livermore National Laboratory, Livermore, CA (November 30, 1992).

4. J. Dennis Lawrence and Warren L. Persons, "A survey of industry practices in developing high-integrity software," Lawrence Livermore National Laboratory, Livermore, CA, in prep.

5. Robert N. Charette, *Software Engineering Risk Analysis and Management*, McGraw-Hill (1989).

6. Frederick P. Brooks, "No silver bullet: essence and accidents of software engineering," *IEEE Computer* **20**, 4 (April 1987), 10–19.

7. "Guide to software quality management system construction and certification," TickIT Project Office, London (February 28, 1992).

8. Nancy G. Leveson and Clark S. Turner, "An investigation of the Therac-25 accidents," *IEEE Computer* **26**, 7 (July 1993), 18–41.

Testing Level	Error Classes						
	Total System Errors	Design Errors	Coding Errors	MMI Errors	Interface Errors	Database Errors	Performance Errors
0	50-70%	~70%	~10%	~15%	~10%	~10%	~5%
1	70-80%	~20%	~75%	~50%	~10%	~50%	~5%
2	80-90%	~5%	~10%	~15%	~50%	~30%	~20%
3	90-95%	~4%	~2%	~15%	~20%	~5%	~50%
4	95-99%		~2%	~4%	~9%	~4%	~19%

Figure 1. Percent of Errors Found by Different Levels of Testing

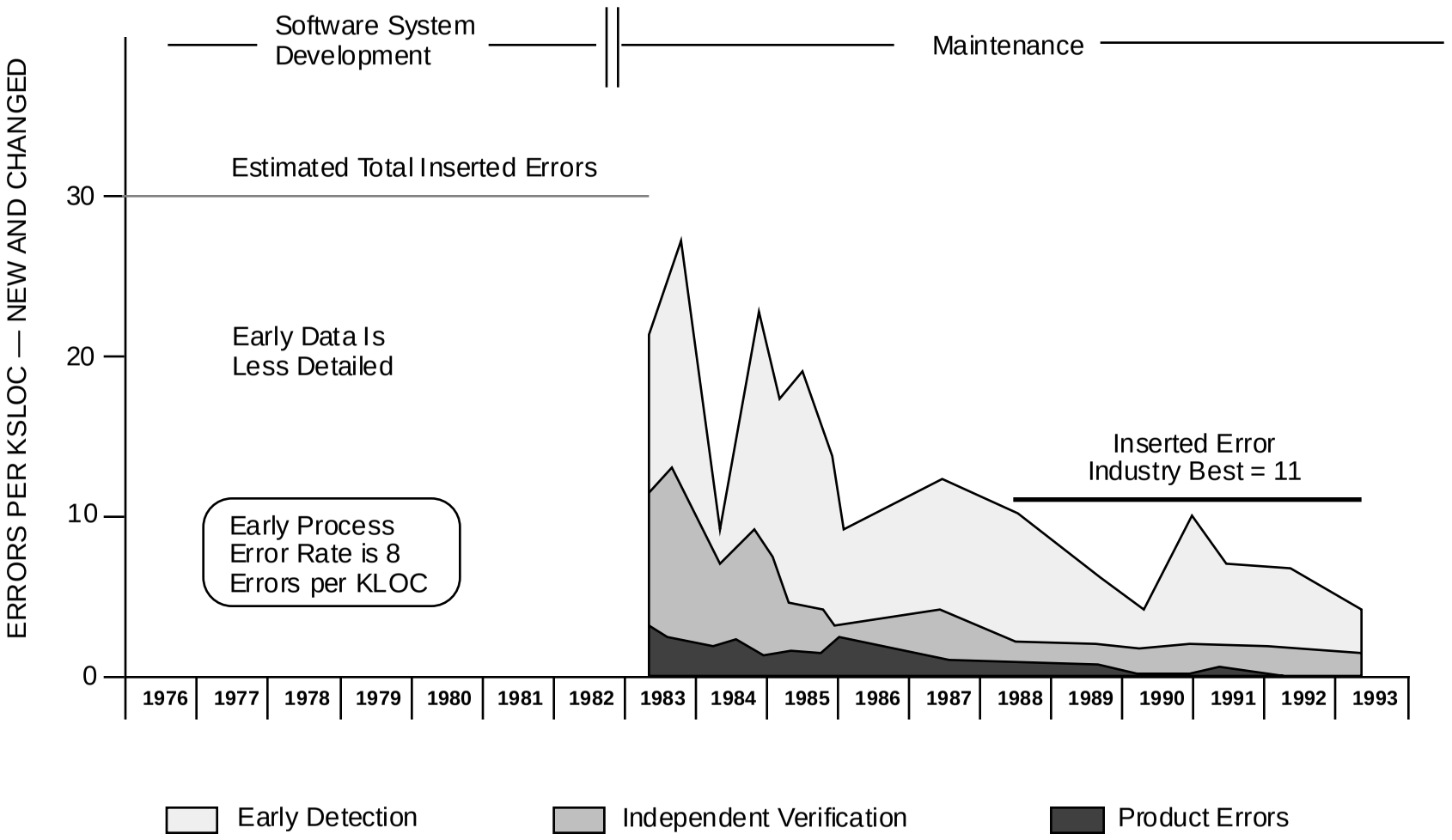


Fig. 2. On-Board Shuttle Software Error Measurements

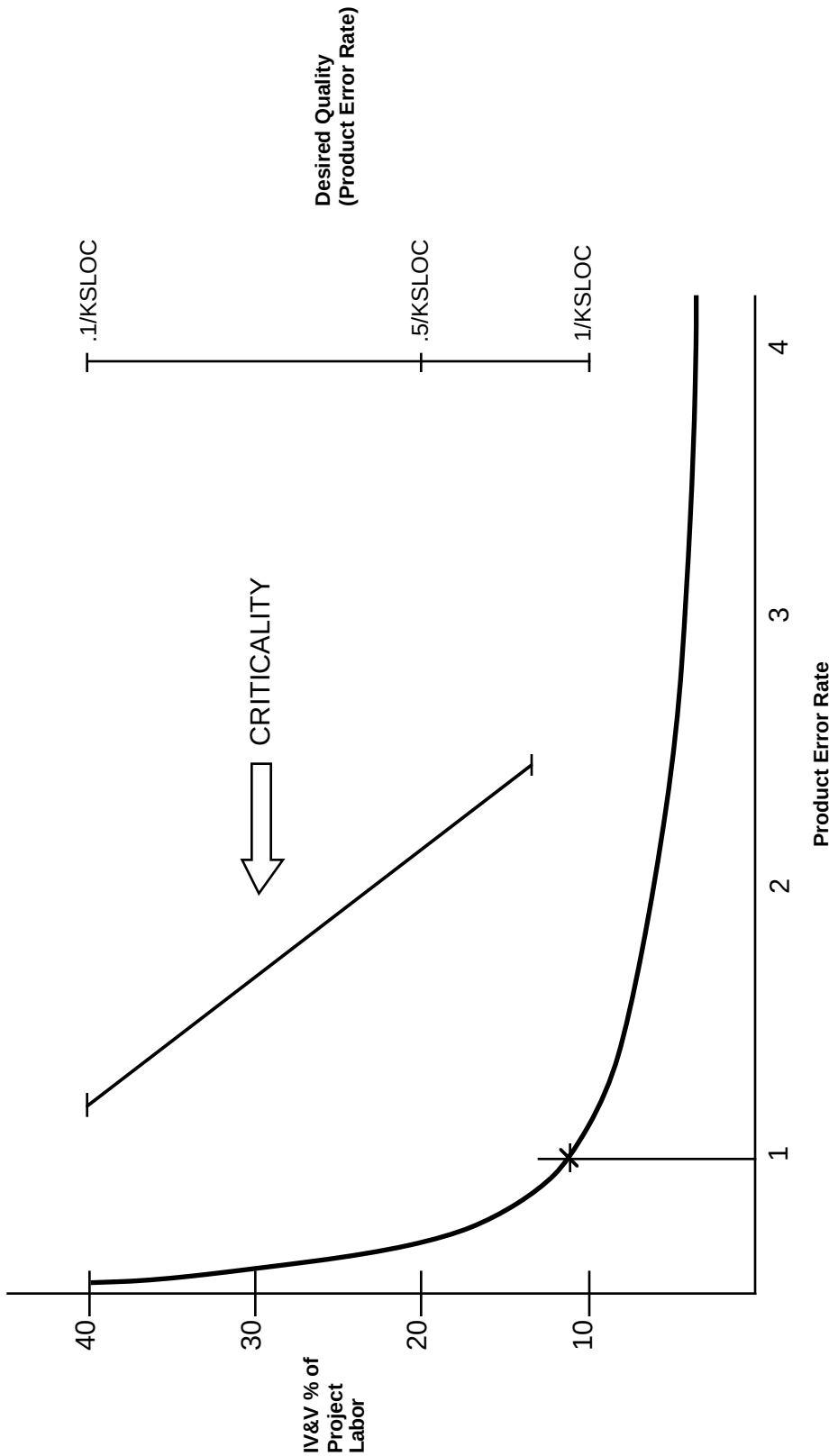


Fig. 3. Software Criticality vs. Cost for Early Development Phase of Large Software Systems

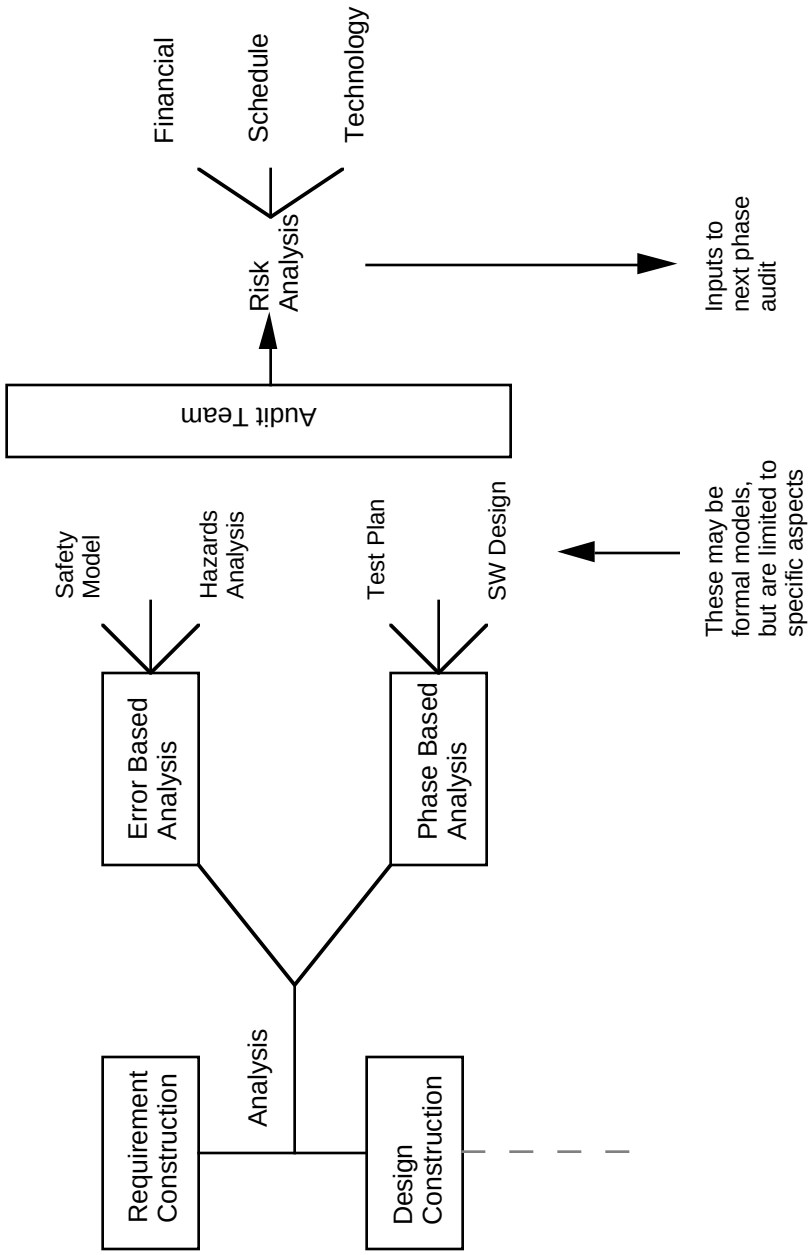


Fig. 4. Life Cycle Phase Construction and Analysis Activities